# Delphi Programmierpraktikum

# Continuo

Höhne, Hermann,
technische Informatik, tinf6626,
zweites Fach- und Verwaltungssemester

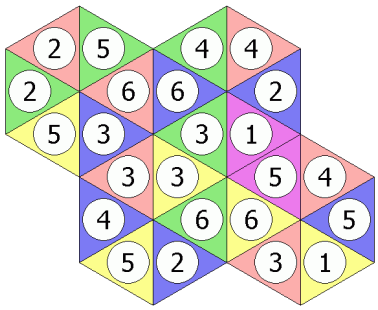
# Inhaltsverzeichnis


1 Benutzerhandbuch....4
1.1 Continuo....4
1.2 Ablaufbedingungen....4
1.3 Programminstallation....5
1.4 Programmstart....5
1.5 Bedienungsanleitung....5
Fenster....5
Spielbeginn....6
Karten legen....6
Spielende....7
Einstellungen....7
1.6 Fehlermeldungen....7
„Karten können nicht gelesen werden: (...)“....7
„Beim Speichern des Kartenstapels ist ein Fehler aufgetreten: (...)“....7
„Beim Speichern der Höchststände ist ein Fehler aufgetreten: (...)“....7
1.7 Wiederanlaufbedingungen....8
2 Programmiererhandbuch....9
2.1 Entwicklungskonfiguration....9
2.2 Problemanalyse und Realisation....9
2.2.1 Problemanalyse....9
Anzeige....9
Spielfeld....9
Bedienung....10
Karten....10
Kartengenerierung....10
Schummelfunktion....10
Dateien....11
2.2.2 Realisationsanalyse....11
Feldstruktur....11
Kartenstapel....11
Bedienung....12
Dateien für Karten und Höchststände....12
Gruppierung....12
2.2.3 Realisationsbeschreibung....13
Sechsecke....13
Anzeige....13
Nachbarschaft....14
Schummelfunktion....14
2.3 Beschreibung grundlegender Datenstrukturen....15
Spielfeld....15
Punkt....15
Karte....16
Anzeige....16
Höchststand....16
2.4 Programmorganisationsplan....17
2.5 Programmtests....17
Bewegen von Karten am Rand....17
Legen einer Karte am Rand....18

Schummeln.....18
Drehen.....18
Laden und Speichern (alle Dateisorten).....18

# 1 Benutzerhandbuch

## 1.1 Continuo




*Continuo* ist ein Solospiel zum Probieren und Rechnen.

Es werden sechseckige Karten aneinandergelegt. Jedes Sechstel einer Karte ist mit einer Farbe und einem Wert versehen. Der Wert 6 kommt auf jeder Karte genau einmal vor. Für jeden Spielzug gibt es Punkte: Mehr je besser die Karten zusammenpassen.

Bei Spielbeginn liegen nur zwei Karten auf dem Spielfeld. Der Spieler erhält die restlichen 30 Spielkarten. Vier davon werden offen hingelegt und können für den nächsten Spielzug verwendet werden. Nach einem Spielzug wird erneut eine der verbliebenen Karten aufgedeckt.

Die Punkte eines Zuges ergeben sich durch folgende Regeln:

- Stimmen Farbe ODER Wert zweier Kanten überein, so werden die Werte der Kanten addiert.
- Stimmen Farbe UND Wert der Kanten überein, so werden die Werte addiert und verdoppelt.
- Legt man die Karte an mehrere Kanten an, so werden die Punkte aus den jeweiligen Kantenübereinstimmungen addiert und mit der Anzahl der Kanten mit Übereinstimmung multipliziert.

Die Punkte eines Spieles ergeben sich aus der Addition der Punkte der Züge.

## 1.2 Ablaufbedingungen

Ihr Rechner benötigt:[1]

Hardware:

- Bildschirm
- Maus (am besten mit Rad)
- Tastatur (optional)
- 12 MB freier Arbeitsspeicher
- Graphikadapter, der mindestens 265 Farben anzeigt
- x68 kompatiblen Prozessor

Software:

- Microsoft Windows ab Windows 95

  oder
- Linux (selbst mit Lazarus Version 0.9.26.2 kompilieren notwendig)

  oder
- wine (getestet mit Version 1.1.25)

1 Getestet auf einem System mit Cyrix M1 Prozessor, 96MB RAM und Win98
wine unter Ubuntu Hardy getestet

## 1.3 Programminstallation

Es ist keine Installation notwendig. Wird *Continuo* in einem Archiv geliefert, entpacken Sie alle Dateien in ein Verzeichnis. *Continuo* kann von jedem Ort ausgeführt werden. Falls Sie *Continuo* auf die Festplatte Ihres Rechners übertragen, kopieren Sie die Datei „Continuo.cto“ wenn möglich mit. Führen Sie *Continuo* in einem Verzeichnis aus, in dem Sie Schreibrechte haben, damit Höchststände gesichert werden können.

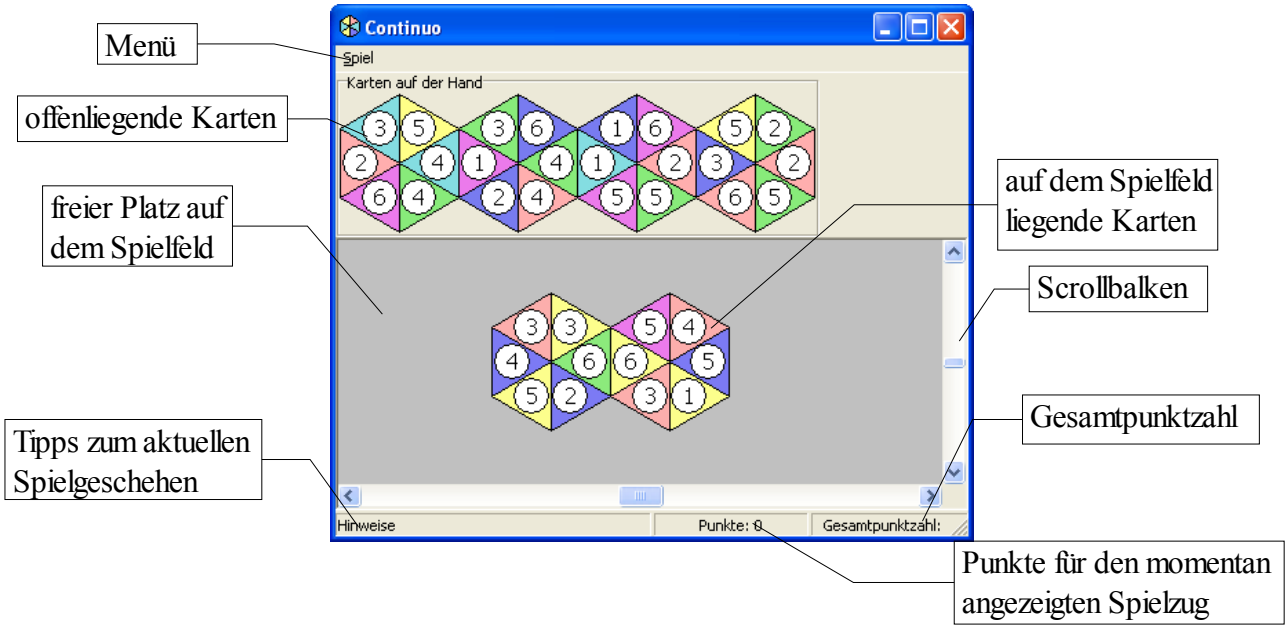
## 1.4 Programmstart

Führen Sie einen Doppelklick auf die ausführbare Datei „Continuo.exe“ aus um das Programm zu starten.

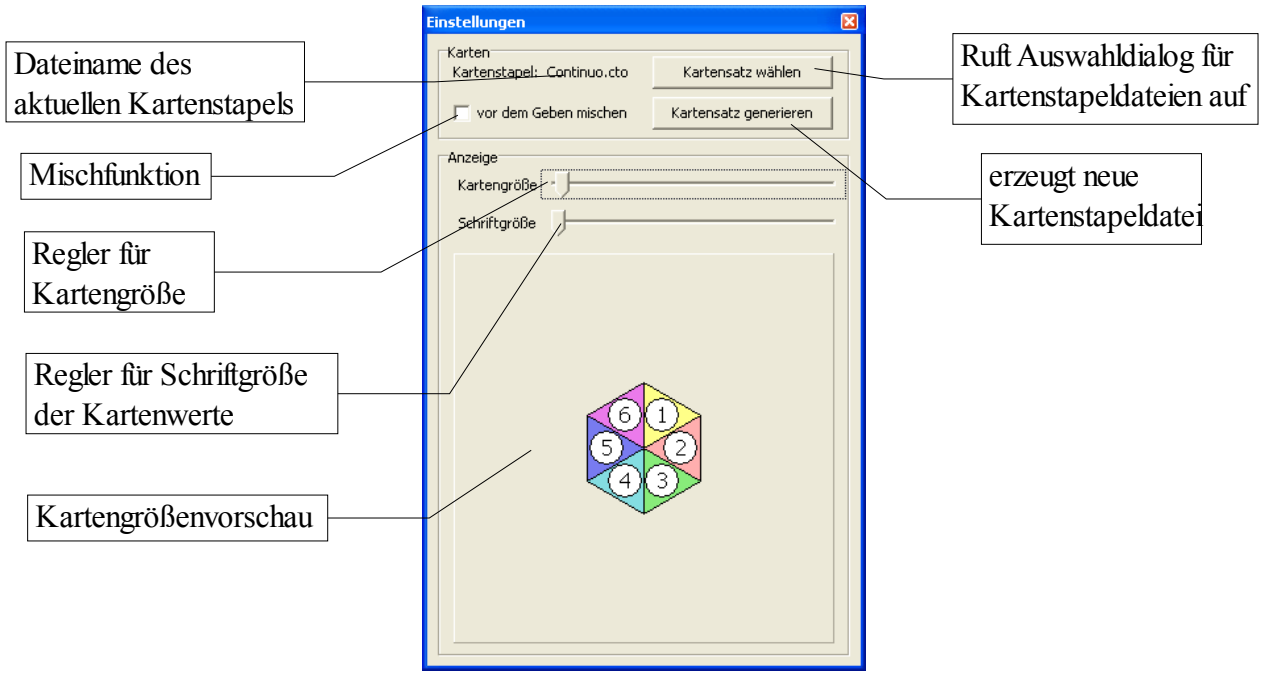
## 1.5 Bedienungsanleitung

### *Fenster*

**Hauptfenster**



**Einstellungen**




**Höchststände**

Das Höchststandsfenster zeigt Ihnen eine Liste an, die Informationen darüber enthält, welcher Spieler an welchem Tag welchen Höchststand mit dem aktuell gewählten Kartenstapel erreicht hat. Nur die zehn besten Endpunktestände von Spielen mit ungemischten Kartenstapeln werden gespeichert.

### *Spielbeginn*

Starten Sie ein neues Spiel, indem Sie mit der Maus den Menüpunkt Spiel→Neu anklicken. Alternativ können Sie die Taste F2 drücken.

### *Karten legen*

Im oberen Bereich des Fensters sehen Sie vier offen liegende Spielkarten. Alle weiteren nicht auf dem Spielfeld liegenden Karten sind unsichtbar. Wählen Sie eine der offen liegenden Karten aus, indem Sie die Karte anklicken.

Bewegen Sie die Maus auf dem Spielfeld. An freien Positionen wird die gerade von Ihnen ausgewählte Karte angezeigt. Drücken Sie die Tasten L oder R oder verwenden Sie das Mausrad, um diese Karte zu drehen.

Klicken Sie auf das Spielfeld, um die Karte so abzulegen, wie sie gerade dargestellt wird. Die Punkte, die Ihnen für diesen Zug gutgeschrieben werden, werden Ihrer Gesamtpunktzahl hinzugefügt. Auch während des Positionierens einer Karte wird der Punktestand in der Statusleiste angezeigt. Ziel des Spieles ist es, einen möglichst hohen Gesamtpunktestand zu erreichen.

Tipp: Wenn Sie keinen guten Zug finden können, drücken Sie die Taste C. Die Schummelfunktion schlägt Ihnen den bestmöglichen Zug vor. Doch Achtung: Schummeln kostet Sie jedes mal 20 Punkte.

### *Spielende*

Das Spiel ist zuende, wenn Sie keine Karten mehr legen können oder Sie das Programm beenden. Haben Sie einen neuen Höchststand erzielt, öffnet sich das Höchststandsfenster automatisch und fragt Sie nach Ihrem Namen.

### *Einstellungen*

Im Einstellungsfenster können Sie Spielverhalten und Anzeigegrößen verändern.

Über den Knopf „Kartensatz wählen“ können Sie einen Kartensatz laden. In einem Dialog werden Sie gefragt, welche Kartenstapeldatei geladen werden soll.
Über den Knopf „Kartensatz generieren“ können Sie einen neuen Kartensatz erzeugen lassen. In dem erscheinenden Dialog wird gefragt, in welche Kartenstapeldatei der neue Kartensatz gespeichert werden soll.
Ferner können Sie bestimmen, ob die Karten vor dem Spielbeginn gemischt werden sollen.
Alle diese Einstellungen verändern das Spielverhalten. Daher wird automatisch ein neues Spiel gestartet, wenn Sie diese Einstellungen bei einem laufenden Spiel verändern.

Die Größe der dargestellten Karten können Sie zu jedem Zeitpunkt ändern. Die neuen Einstellungen werden sofort, auch bei laufenden Spielen, übernommen.

## *1.6 Fehlermeldungen*

### *„Karten können nicht gelesen werden: (...)“*

**Ursache:** *Continuo* liest Kartenstapel aus Dateien mit der Endung .cto. Im Normalfall versucht *Continuo* beim Spielstart die Datei „Continuo.cto“ zu öffnen. Kann diese Datei (oder die Datei, welche Sie in den Einstellungen gewählt haben) nicht gelesen werden, so erscheint o.g. Fehlermeldung.

**Maßnahme:** Wählen Sie im Einstellungsmenü eine andere Kartenstapeldatei aus. Falls Sie über keine Kartenstapeldatei verfügen, können Sie über den Knopf „neuen Kartenstapel erzeugen“ einen Stapel zufällig generierter Karten in eine Datei speichern lassen und diese verwenden.

### „Beim Speichern des Kartenstapels ist ein Fehler aufgetreten: (...)“

**Ursache:** Die von von *Continuo* erzeugten Karten können nicht in die von Ihnen gewählte Datei gespeichert werden. Dies ist u.a. der Fall, wenn Ihnen die zum Schreiben der Datei nötigen Rechte fehlen, das Laufwerk schreibgeschützt ist oder der Pfad nicht gültig ist.

**Maßnahme:** Wählen Sie in Einstellungsmenü einen anderen Ort zum Speichern des Kartenstapels.

### „Beim Speichern der Höchststände ist ein Fehler aufgetreten: (...)“

**Ursache:** Der Punktstand, den Sie zuletzt erreicht haben, konnte nicht in der Datei für Höchststände abgespeichert werden. Diese Datei hat den gleichen Namen wie der ausgewählte Kartenstapel, jedoch die Dateiendung „.cts“. Dies ist u.a. der Fall, wenn Ihnen die zum Schreiben der Datei nötigen Rechte fehlen, das Laufwerk schreibgeschützt ist oder der Pfad nicht gültig ist.

**Maßnahme:** Entfernen Sie den Schreibschutz von der Datei. Alternativ können Sie einen anderen Kartenstapel wählen, sodass die Datei für Höchststände neu angelegt oder überschrieben werden kann.

## *1.7 Wiederanlaufbedingungen*

Wird das Programm unterbrochen, kann der letzte Spielstand nicht wiederhergestellt werden. Das Festhalten eines Spielzustandes ist generell nicht möglich.

# 2 Programmiererhandbuch

## *2.1 Entwicklungskonfiguration*

Software:

- Programmierung
  - „Borland Delphi 2007 for Win32“ (Web Installer vom September 2008)
  - „Turbo Delphi“ (Borland Delphi 2006)
  - Lazarus Version 0.9.26.2
- IrfanView, MSPaint und GIMP zum Zeichnen des Anwendungssymbols

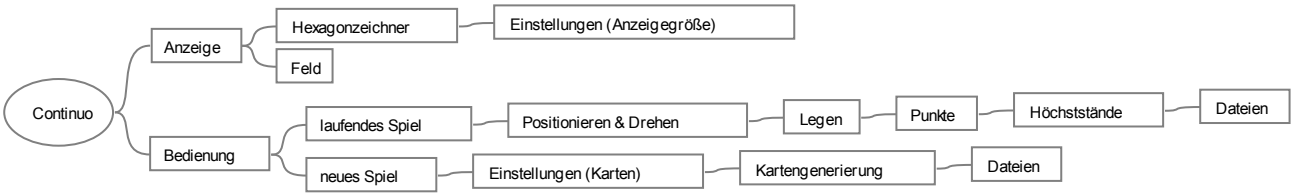
Hardware:

- Asus eeePC 1000H (Windows XP Home SP 3):
  Intel Atom CPU
  1 GB RAM
- Hermanns Desktop Rechner (Windows XP Professional 64bit SP 2):
  AMD Athlon X2 Dual Core 5000+
  ASRock AliveXFire eSATA2 Mainboard
  4 GB RAM

## *2.2 Problemanalyse und Realisation*

### 2.2.1 Problemanalyse

Während der Entwicklung wurden die Anforderungen problemorientiert in dieser Reihenfolge analysiert:




#### *Anzeige*

Da ein Spieler auch alle 32 Karten in einer Reihe aneinander legen darf, muss das Programm die Möglichkeit haben, nur den wichtigen Teil des Spielfeldes anzeigen zu können und den Rest auszublenden, damit die Karten auch bei geringer Bildschirmauflösung noch akzeptable Größen haben. Eine bei Fenstergrößenveränderung skalierende Anzeige wird von kaum einem Benutzer erwartet. Fast ohne Ausnahme verschafft sich der Benutzer durch das Maximieren eines Fensters mehr Übersicht. Dies soll auch in *Continuo* der Fall sein. Die Größe der Karten soll zwar fest, aber vom Benutzer bestimmbar sein.

#### *Spielfeld*

Die Aufgabe verlangt eine gute Planung der Datenstruktur des Spielfeldes. Ist sie günstig gewählt,

wird die Realisierung des restlichen Programmes um vieles vereinfacht. Besonders wichtig für die Spiellogik sind Nachbarschaftsbeziehungen. Das Spielfeld ist ein Wabenmuster. Es gibt, von einer Position ausgehend also sechs Nachbarfelder.

### *Bedienung*

Das Programm ist ein Spiel, es soll also für jeden Benutzer bedienbar sein. Die Oberfläche soll daher möglichst einfach zu verstehen sein. Das Positionieren einer Karte mit der Maus und das Ablegen mittels Mausklick ist intuitiv verständlich. Das Drehen der Karte mit dem Mausrad ist praktisch, da der Spieler dann nur die Maus verwenden muss. Das Spiel kann also Einhändig gespielt werden, was gerade für ein Gelegenheitsspiel wie *Continuo* gut ist.[2] Hat die verwendete Maus kein Mausrad, muss zum Drehen eine alternative Steuerung über Tasten oder das Menü, angeboten werden. Hinweistexte in der Statusleiste enthalten während des Spieles nützliche Hinweise, besonders für die Erstbenutzung.

### *Karten*

Die Verwaltung der verdeckten Karten und der Karten auf der Hand (damit sind die vier sichtbaren Karten gemeint) ist relativ einfach. Liegen die Karten in sortierter Form vor, so muss das Programm nur die ersten vier Karten für den Spieler „ziehen“ und anzeigen. Dies ist sich nicht anders, wenn die Karten zufällig „gezogen“ werden sollen. Das Programm muss lediglich die Karten vorher mischen und die ersten vier Karten der daraus entstehenden Permutation anzeigen. Eine dynamische Liste zum verwalten der Karten ist günstiger als ein statisches Array, da das Entfernen einer beliebigen Karte ohne Umschichten aller folgenden geschehen soll.

### *Kartengenerierung*

Sind keine Spielkarten in einer Datei mitgeliefert, muss das Programm selbst welche erzeugen können. Die Karten sollen zufällige Werten und Farben haben, müssen jedoch mit den Continuo-Regeln konform sein. Das erstellen einer Karte hat somit drei Schritte:

Im ersten werden Farben und Werte vollkommen zufällig gewählt.

Im zweiten Schritt wird die Karte auf Gültigkeit überprüft. Wird eine Regel gebrochen (liegen beispielsweise zwei gleiche Farben nebeneinander), muss die Karte an dieser stelle korrigiert werden (durch Überschreiben mit einer anderen zufällig gewählten Farbe). Nach dieser Maßnahme kann der gleiche Fehler wieder auftauchen, notfalls an anderer stelle auf der Karte. Der zweite Schritt läuft daher oft wiederholt. Die Wiederholung ließe sich vermeiden, wenn das Programm den Fehler auf der zufälligen Karte analysiert und aus den verbleibenden Möglichkeiten, eine Farbe oder einen Wert an einer gültigen Position einzufügen, eine auswählt. Jedoch wäre die intelligente Auswahl aufgrund der erhöhten Komplexität vermutlich nicht schneller als das einfache Probieren.

Im dritten Schritt werden finale Korrekturen vorgenommen, wie beispielsweise das Hinzufügen einer Sechs, falls nötig.

### *Schummelfunktion*

Ein Brute-Force-Algorithmus ist an dieser Stelle sowohl simpel als auch effektiv. Das Programm muss lediglich folgende Schritte ausführen:

- alle möglichen Züge ausprobieren
- die Punkte dieser Zügen berechnen

2 Auch wenn es in der FH-Wedel verboten ist, im Rechenzentrum am PC zu essen.

- Zug mit dem höchsten Punktestand auswählen

Es ist möglich, dass es Ressourcen sparendere, vielleicht sogar deterministische Algorithmen hierfür gibt. Außerdem kann es sein, dass der letztlich gewählte Ansatz selbst unter den Brute-Force-Methoden außerordentlich viel Speicher verwendet. Zur Effizienzsteigerung ist es sinnvoll, im ersten Schritt nicht tatsächlich alle Züge zu betrachten, sondern zunächst zu filtern: Karten können nicht irgendwo auf das Feld gelegt werden, sondern nur an bereits liegende Karten angelegt werden. Damit fällt schon ein Großteil der Züge weg. Außerdem können Karten nicht aufeinander liegen.

#### *Dateien*

Das Programm muss nur an zwei stellen mit Dateien arbeiten: Beim Kartenstapel und bei den Höchstständen. Vor dem einlesen einer Datei wird als sehr einfacher Dateityptest ihre Länge geprüft. Bei Karten kann die Funktion zur Gültigkeitsprüfung, welche zur Kartengenerierung gebraucht wird, als zusätzliche Prüfung geschehen. Es können fehlerhafte Karten in das Spiel gelangen, wenn die Farben in den Karten nicht zur Palette (bzw. den Farben in den Konstanten) passen. Höchststände könnten nur durch das Anhängen von Prüfsummen beim speichern gegen Veränderung durch den Benutzer geschützt werden.

Bei Fehlern sollen die unteren Datenhaltungsunits Exceptions auslösen, die erst in den Formularunits verarbeitet werden. Wäre eine Lokalisierung des Programmes geplant, müssten dann nur die Formularunits geändert werden, da in den unteren Units kein benutzerrelevanter Text steht.

### 2.2.2 Realisationsanalyse

#### *Feldstruktur*

Die erste Idee war eine komplett dynamische Struktur: Eine sechsfach verkettete Liste in Netzstruktur, die sich selbst sofort beim Einfügen eines Datensatzes die Nachbarschaftsbeziehungen durch geschicktes Umbiegen der Zeiger merkt. Das Implementieren von Nachbarschaften, die entstehen, wenn der Spieler Ringe legt, schien nur durch Rekursionen möglich zu sein. Diese müssten aber sehr tief laufen können. Wegen der daraus möglicherweise entstehenden Probleme (Stack Overflow) wurde diese Idee verworfen.

Die zweite und verwendete Idee ist eine halbdynamische Lösung. Obwohl die Waben zwar ein Sechseckmuster haben, so können die Feldpositionen doch von einer einfachen zweidimensionalen Struktur repräsentiert werden, wenn man nur einfach jede zweite Zeile (in diesem Fall) um eine halbe Position verschiebt. Es ist jedoch nicht Sinnvoll, auf jeder Position des Spielfeldes Karteninformationen abzulegen, da das gesamte Feld nie belegt sein wird.
Dieser Ansatz führte zu einem statischen Array von Pointern auf Karten.

Eine Positionen auf dem Spielfeld wird immer durch die Zeilen- und Spaltenangabe für den Arrayindex des Spielfeldes angegeben. Um eine Position einfacher von Funktionen zurückgeben zu lassen, ist es sinnvoll, dafür einen eigenen Datentyp zu definieren.

#### *Kartenstapel*

Der Kartenstapel sollte durch eine dynamische Liste repräsentiert werden. Der Index einer Karte sei durch ihre Position in der Liste gegeben, ist also durch Mitzählen beim Durchlaufen der Liste bestimmbar.

Ein statisches Array mit 32 Plätzen ist auch denkbar. Allerdings müssen, wenn der Spieler eine

Karte aus der Hand (also eines der ersten vier Elemente aus der Liste), die folgenden Elemente nachrücken, was nicht sehr elegant ist.

### *Bedienung*

Da nie das ganze Spielfeld angezeigt wird, muss der sichtbare Bereich kleiner als das Spielfeld sein. Hier kommt dem Projekt die Delphi-Komponentenbibliothek entgegen: Eine TScrollBox bietet genau die benötigte Funktionalität. Ein Großer innerer Bereich kann in einem kleineren Sichtbereich verschoben werden. Scrollen ist im Allgemeinen eine akzeptierte und gern gesehene Programmfunktion.[3]

Das Spielfeld soll durch eine Bitmapgraphik dargestellt werden. Hier bieten sich eine TPaintBox oder auch das TBitmap innerhalb eines TImage an. Eine TPaintBox benötigt weniger Hauptspeicher, da diese Komponente ihren Inhalt nicht speichert. Dies wird beim Scrollen jedoch zum Nachteil: Verschwindet ein bereits gezeichneter Bereich aus dem sichtbaren Bereich, so muss er wieder neu gezeichnet werden; das Zeichnen über TCanvas ist aber relativ langsam und kann durch das Speichern des gezeichneten in einem TBitmap sehr einfach gelöst werden.

Nachteil der Bitmap ist allerdings, dass ihre Größe vorher gesetzt werden muss und auch die leeren Bereiche Speicher verbrauchen. Daher ist es sinnvoll, die Größe der Bitmap so klein wie möglich zu halten. Dazu müssen nicht nur die Ausdehnung der liegenden Karten gefunden werden, sondern auch die ihrer Nachbarfelder, damit auch an den Positionen, an denen der Benutzer eine Karte legen wollen könnte, das Darstellen einer Karte möglich ist. Damit beim Neusetzen der Bitmapgröße beim Legen einer Karte das Spielfeld nicht in der Ansicht umherspringt, wird das anzeigende TImage auf ein TPanel mit der immer gleichen maximalen Spielfeldgröße gelegt. Auf dem TPanel wird das TImage mittels der Positionseigenschaften Left und Top positioniert.

Das Einstellungsmenü ist die einzige Stelle, an welcher der Benutzer Eingaben machen kann, die Fehlerhaft sein können. Aus diesem Grund wird für die Steuerung der Anzeigegröße der Karten eine TTrackBar verwendet, da diese keine fehlerhaften Eingaben zulässt.

### *Dateien für Karten und Höchststände*

Bei den Höchstständen ist das kaum anders, denn hier ist das Einfügen eines neuen Datensatzes an beliebiger Position einfach. Aufgrund der genau begrenzten Anzahl von Einträgen wäre eine statische Lösung allerdings auch gut möglich.

### *Gruppierung*

Graphische und logische Funktionen sind strikt getrennt. Lediglich in den Formularunits ist diese Trennung aufgehoben, da zwangsläufig die gesamte Funktionalität zur Verfügung stehen muss. Dafür sollen die Funktionen Formularunits so kurz wie möglich gehalten werden und nach Möglichkeit nur aus Einzeilern bestehen.

Alle Funktionen der Datenhaltung (die dynamischen Listen) sollen in eigenen Units ruhen.

Eine Kapselung ist wünschenswert, bläht die in der Hierarchie relativ weit oben stehenden Units unangenehm auf. Das möglichst lokale Definieren von Hilfsfunktionen führt an einigen Stellen fast zur Codeobfuskation. Daher werden, weil es Pflicht ist, die Hilfsfunktionen erst kurz vor dem Projektabschluss in die Deklarationsteile der aufrufenden Funktionen verschoben.

3 Das Pseudo-Scrollen durch Tastendruck, welches beispielsweise auf Smartphones eingesetzt wird, wird auf Dauer sehr anstrengend.

## 2.2.3 Realisationsbeschreibung

### *Sechsecke*

Die Funktionen zum Zeichnen von sechseckigen Karten werden selbst geschrieben und bekommen eine eigene Unit. Das Zeichnen eines Sechsecks an eine beliebige Position auf einer graphischen Ausgabeeinheit ist eine allgemein verwendbare Funktion: Beim Zeichnen an sich ist es egal, ob es sich um eine Karte auf der Hand oder auf dem Feld handelt.

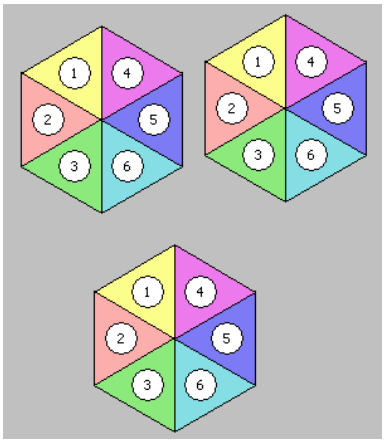


*Abbildung 1: Der Kartenzeichner in einem frühen Programmtest.*

Beim Zeichnen werden die sechs Teildreiecke des Sechseckes gemalt. Die Füllfarbe wird durch die Karteninformation bestimmt. Der Zeichenort des Sechsecks wird durch die Position seines Mittelpunktes bestimmt. Das Aufzeichnen der sechs Werte wird aufgrund des semantischen Zusammenhangs in derselben Unit implementiert. Die Größe einer Karte wird durch die Angabe des vertikalen Durchmessers des Sechsecks in Pixeln definiert. Die Höhe der Teildreiecke wird dementsprechend berechnet.

Das tatsächliche Zeichnen eines farbigen Dreiecks wird mit der Prozedur `Polygon`, welche in `TCanvas` definiert ist, realisiert. Zur Anzeige der Nummern auf einem Kreis werden `TextOut` und `Ellipse` derselben von Borland mitgelieferten Klasse verwendet.

Bei Programmtests wurde festgestellt, dass die Kanten der Karten auf dem Feld bei bestimmten Kartengrößen außeinanderlaufen. Die Eckpunkte liegen also nicht exakt aufeinander. Dies wird durch Rundungsfehler bei der Berechnung der Eckpunkte der Teildreiecke verursacht. Erst bei einem Test der Schummelfunktion fiel auf, dass die in Abbildung 1 zu sehenden Werte in ost-west-Richtung gespiegelt werden müssen und die Darstellung der Karte insgesamt auf dem Kopf steht.

### *Anzeige*

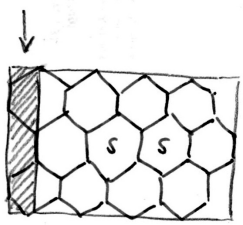


*Abbildung 3: Startkarten in gerader Zeile*

Die Anzeige des Spielfeldes soll durch eine Bitmapgraphik geschehen. Da sich die Größe des hierfür gewählten TBitmap nicht dynamisch anpasst, muss sie vorher berechnet werden.

Die Berechnung der Breite der Anzeige ist simpel und einfach nachzuvollziehen: Es muss genug Platz für alle Spalten des Feldes geben. Jede zweite Zeile ist in der Anzeige um eine halbe Kartenbreite nach rechts verschoben, dieser Platz muss also auch vorhanden sein.

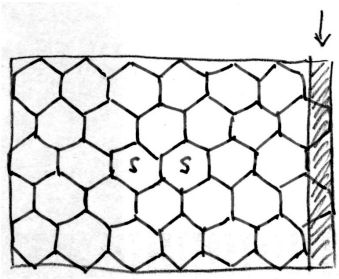


*Abbildung 2: Startkarten in ungerader Zeile*

```
bitmapWidth := cColumns * hexWidth
             + hexWidth DIV 2
             + 1;
```

Die Berechnung der Höhe gestaltet sich etwas komplizierter. Da die Feldgröße von der Anzahl der Maximal in einer Richtung aneinanderlegbaren Karten begrenzt ist, ist die Zahl der Feldzeilen immer Ungerade. Eine gerade Anzahl von Zeilen lässt hingegen sich durch ein Vielfaches von zwei Zeilen ausdrücken. Dadurch dass die sechseckigen Karten aus gleichseitigen Teildreiecke bestehen, ist die Höhe von jeweils zwei Zeilen genau das eineinhalbfache der Kartenhöhe. Für die letzte Zeile

muss lediglich noch eine Kartenhöhe hinzugefügt werden. Dies ist am einfachsten nachvollziehbar, wenn man von einer oberen Kartenspitze ausgeht und die Anzahl der Seitenlängen der Teildreiecke bis zum unteren Feldende zählt.

```
bitmapHeight := ((cRows-1) DIV 2) * (hexHeight * 3 DIV 2)
              + hexHeight
              + 1;
```

Aus Gründen die unbekannt sind, ist bei einer derartigen Initialisierung die Bitmap traditionell immer um einen Pixel an jeder Seite zu kurz, daher die Erhöhung um 1.

Bei dieser Berechnung kommt es leider zur Verschwendung von Platz auf der Bitmap. Wie in Abbildung 3 anhand des schraffierten Bereiches zu erkennen wird am Rand platz für die Anzeige von Feldplätzen reserviert, die nie belegt sein können, weil die Menge von legbaren Karten nicht ausreicht. Diese Platzverschwendung wird ausgeglichen, indem die Anzeigegröße auf den tatsächlich benötigten Bereich verringert wird. Diese Maßnahme und das Reduzieren der Farbtiefe des Bitmaps spart viel Arbeitsspeicher.

Die verwendete Komponente TScrollBox erweist sich als widerspenstig. Die Eigenschaft „Position“ eines Scrollbalkens repräsentiert nicht die Position des angezeigten inneren Bereichs sondern tatsächlich die Position des Scrollbalkens auf der Scrollleiste. Diese Position ist in der normalen Anwendung jedoch scheinbar belanglos. Nichtsdestoweniger soll die Anzeige bei Spielbeginn auf die Spielfeldmitte springen, damit der Benutzer die beiden Startkarten sieht. Zur Bestimmung der Scrollbalkenposition an welcher die Ansicht zentriert ist, wird dem Scrollbalken ein unerreichbar großer Wert zugewiesen, sodass er an seine Maximalposition springt. Die Hälfte des aktuellen Positionswertes repräsentiert nun effektiv die mittlere Position.

### *Nachbarschaft*

Sehr eng verbunden mit der Wahl der Datenstruktur für das Spielfeld ist die hier beschriebene Ermittlung von Nachbarn auf dem Feld. Wie in Abbildung 4 verdeutlicht, liegt an der östlichen Kante einer Karte an Feldposition `(x,y)` immer die westliche Kante der Karte an Feldposition `(x+1,y)`, in nordöstlicher Richtung liegt Karte `(x+1,y-1)` und so weiter.

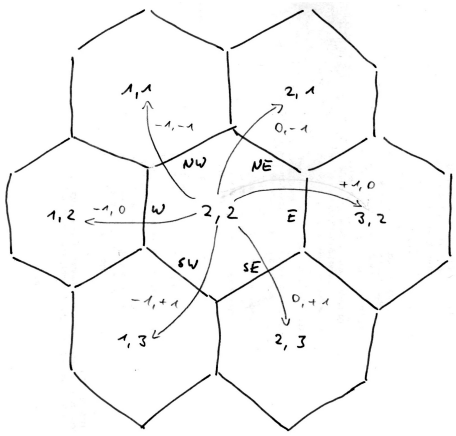



*Abbildung 4: Nachbarschaft in Zeilen mit geradem Index.*

Realisiert wird diese Funktion durch eine Unterscheidung der sechs Nachbarn einer Karte mittels `case`.

Beim Test der Schummelfunktion fiel auf, dass die Nachbarschaftsbeziehungen für Zeilen gerade und ungeradem Index verschieden sind, wie im Vergleich der beiden Abbildungen zu erkennen.

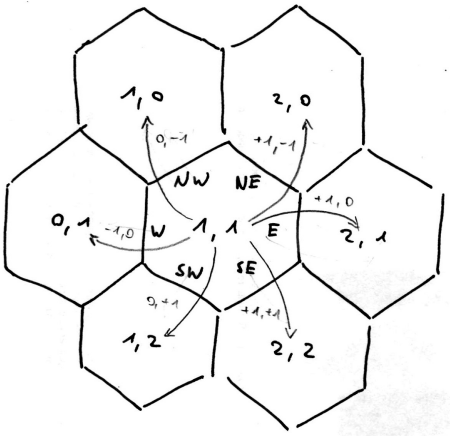



*Abbildung 5: Nachbarschaft in Zeilen mit ungeradem Index.*

### *Schummelfunktion*

Die Schummelfunktion wird durch einen einfachen Brute-Force-Algorithmus realisiert. Durch Iterationen werden alle Feldpositionen durchlaufen, falls der Spieler dort überhaupt eine Karte legen kann, probiert der Algorithmus alle Karten auf der Hand aus (ohne sie aus der Hand zu entfernen). Dabei werden die Punkte, die ein Legen von jeweils einer Karte in jeder Ausrichtung (also insgesamt sechs mal gedreht) an eine Position

mit sich brächte, berechnet und zusammen mit der Information der Ausrichtung in Form der Kartendaten und der Position der Karte auf der Hand in Form des Kartenlistenindexes gespeichert. Hier die Iterationen als Umriss:

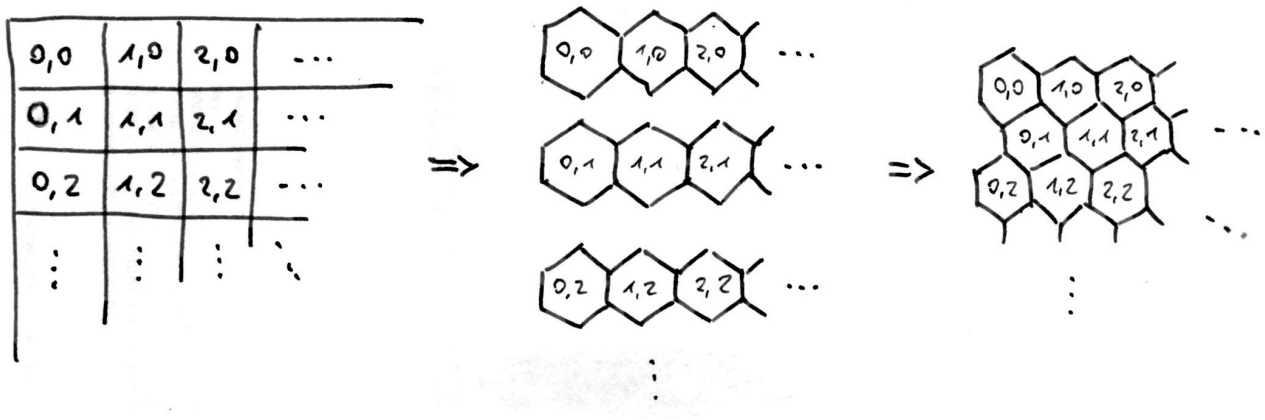
```
for r := low(r) to high(r) do
  for c := low(c) to high(c) do
    if not FieldPositionOccupied(pos) and HasNeighbor(pos) then
      for i := low(i) to high(i) do
        for d := 0 to 5 do
```

Die ersten beiden Schleifen gehen durch alle Feldpositionen, die vorletzte durch alle Karten auf der Hand, die letzte dreht die Karte. Den genauen Algorithmus finden Sie in `UCheater.pas`.

## 2.3 Beschreibung grundlegender Datenstrukturen

### Spielfeld

Das Spielfeld wird durch ein zweidimensionales statisches Array von Pointern auf Kartendaten repräsentiert.




```
type
  TCardPointer = ^TcardProp;
  TField = array[TFieldColumns,TFieldRows] of TcardPointer;
```

Über den Index des Arrays kann auf jede einzelne Positionen des Spielfeldes zugegriffen werden. Liegt an einer Position keine Karte, ist der Inhalt der Arrayzelle `nil`.

Dieser Datentyp kommt nur in der einzigen unitglobalen Variable in `UField` vor. Über die Schnittstellen können andere Units Karten auf die Positionen legen (schreiben) oder Karteninformationen von einer Position lesen. `UField` verhindert dabei, dass Karten gestapelt oder leere Felder als Karte gelesen werden.

### Punkt

```
type
  TFieldPosition = record
    col : TFieldColumns;
    row : TFieldRows;
  end;
```

Eine Position auf dem Spielfeld wird durch die Angabe seiner Zeile und Spalte gegeben. Die meisten Funktionen berechnen aber Positionen. Es ist daher sinnvoll, eine Positionsangabe in einem Typ zu gruppieren.

Stellenweise wird auch der vordefinierte Punkttyp `TPoint` als Positionsangabe verwendet. Dieser kommt entweder bei Bitmappositionen zum Einstatz oder dann, wenn nicht klar ist, ob eine errechnete Position überhaupt eine gültige Spielfeldposition ist.

### *Karte*

```
Type
  TEdge   = (edNE, edE, edSE, edSW, edW, edNW);
  TValue  = 1..cMaxValue;
  TValues = array [TEdge] of TValue;
  TColors = array [TEdge] of TColor;
  TCardProp = packed record
    Values: TValues;
    Colors: TColors;
  end;
```

Implementation des vorgegebenen Spielkartentyps. Jedes Sechstel der sechseckigen Karte kann durch die Angabe ihrer Kante adressiert werden. Jedes Sechstel kann eine individuelle Farbe und einen Wert beinhalten.

```
type
  TCardListNodePointer = ^TCardListNode;
  TCardListNode = record
    card : TCardProp;
    next : TCardListNodePointer;
  end;
```

Typdefinition für eine einfach verkettete Liste zum festhalten von Karten.

### *Anzeige*

```
type
  THexPoints = array [0..6] of TPoint;
  TTriangle = array[0..3] of Tpoint;
```

Der Rahmen eines Sechsecks (oder Dreiecks) kann durch seine Eckpunkte beschrieben werden. Der zusätzliche Punkt des Arrays sorgt dafür, dass der Rahmen durch `Polyline` geschlossen gezeichnet wird.

### *Höchststand*

```
type
  TScore = packed record
    name : shortstring;
    date : TDateTime;
    score : integer;
  end;
```

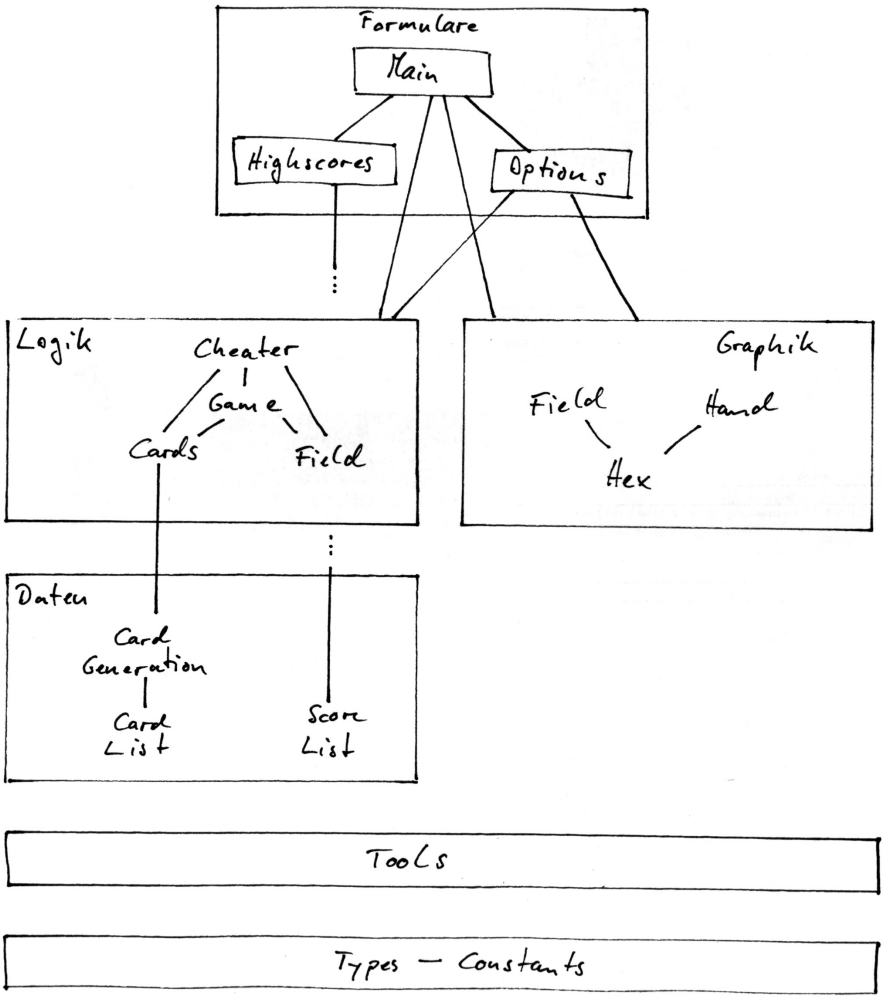
Einfacher record zum Speichern von Höchstständen.

```
type
  TScoreListNodePointer = ^TScoreListNode;
  TScoreListNode = record
    score : TScore;
    next : TScoreListNodePointer;
  end;
```

Typdefinition für eine einfach verkettete Liste zum festhalten von Höchstständen.

## 2.4 Programmorganisationsplan

Diese Graphik verdeutlicht das Ineinandergreifen der nach verschiedenen Anwendungsgebieten gruppierten Units:




Zur Wahrung der Übersichtlichkeit sind hier nicht alle Abhängigkeiten eingetragen. Fast alle Units verwenden Typen und Konstanten.

## 2.5 Programmtests

Diese Tests zeigten während der Entwicklung Fehler auf. Eine Funktion wurde immer direkt nach ihrer Implementierung getestet. Alle hier angesprochenen Probleme wurden nach Ermittlung der Ursache gelöst. Weitere Probleme sind nicht bekannt.

### *Bewegen von Karten am Rand*

**Testfall:** Im laufenden Spiel wird eine Karte ausgewählt. Der Mauszeiger wird zum Rand des Spielfeldes gesteuert.
**erwartetes Ergebnis:** Die „schwebende“ Karte wird nur an gültigen Positionen angezeigt.
**zunächst erzieltes Ergebnis:** Die „schwebende“ Karten wird auch außerhalb des Spielfeldes gezeichnet und ist dort nicht oder nur teilweise zu sehen.
**Ursache:** In keiner der beteiligten Routinen war dies verboten worden.

### *Legen einer Karte am Rand*

**Testfall:** Im laufenden Spiel wird eine Karte ausgewählt und durch einen Klick an einer der Spielfeldränder abgelegt.
**erwartetes Ergebnis:** Die Karte wird abgelegt.
**zunächst erzieltes Ergebnis:** Es tritt ein Bereichsprüfungsfehler auf.
**Ursache:** Die Funktion `FindNeighbor` erzeugte „virtuelle“ Nachbarn mit Indizes, die außerhalb des Wertebereiches des Feldes lagen.

### *Schummeln*

**Testfall:** Im laufenden Spiel wird die Schummelfunktion aufgerufen.
**erwartetes Ergebnis:** Die Karte, welche geschickt angelegt die größte Punktzahl im nächsten Zug bringt, wird gedreht. Die Karte und ihre ideale Position auf dem Feld wird markiert.
**zunächst erzieltes Ergebnis:** Die Schummelfunktion schlägt das Ablegen an scheinbar unmöglichen Positionen vor.
**Ursache:** Die Kartenzeichenfunktion `DrawCard` arbeitete fehlerhaft, sodass sich das sichtbare Spielfeld vom tatsächlichen (logischen) unterschied.

### *Drehen*

**Testfall:** Im laufenden Spiel wird eine Karte ausgewählt und abgelegt. Der Spieler verwendet anschließend das Mausrad oder die Tastatur, um die Karte zu drehen.
**erwartetes Ergebnis:** Nichts geschieht.
**zunächst erzieltes Ergebnis:** In der Anzeige wird die bereits liegende Karte wird gedreht.
**Ursache:** Es wurde vor dem Drehen nicht geprüft, ob die Karte bereits abgelegt wurde.

### *Laden und Speichern (alle Dateisorten)*

**Testfall:** Eine nicht existierende Datei soll gelesen werden, eine schreibgeschützte Datei soll überschrieben werden, eine Datei hat ungültige Länge, eine Datei enthält ungültige Daten (nur Karten) oder der Handle zu einer Datei wird während der Operation geschlossen.
**erwartetes Ergebnis:** Eine entsprechende Fehlermeldung wird ausgegeben.
**erzieltes Ergebnis:** Eine entsprechende Fehlermeldung wird ausgegeben.